Exam 1 – Notes

**Dynamic Programming**

1. Describe Table Entry in Words

* T(i) = [Reword of problem] in a[1]…a[i]
* T(i) = [Reword of problem] in a[1]…a[i] which includes a[i]
* T(i) = [Reword of problem] in a[1]…a[i] at i
* T(i) = [Reword of problem] in a[1]…a[i] ending at a[i]

1. Define Recurrence of the Subproblems

* Include the Base Case
* Include the Range
* Index into the inputs from 1
* Use Mathematical Notation

1. Pseudocode

* Language Agnostic
* Use of language-specific functions/notation likely to be penalized
* Index into the inputs from 1

1. Runtime Analysis

* In Big O Notation – O()
* All non-trivial parts – slower then O(1)

**Longest Increasing Subsequence**

1. If your problem only has a single array
2. If your problem is comparing to itself
3. If your problem only looks back one element at a time (not a window, or bag)
4. This is normally (there are exceptions) a 1-D table
5. Two variations:

* With O(n) lookback:
  + If you need to check all previous elements
  + If you need to check all previous elements > or < some requirement
  + This results in O(n2)
* With O(1) lookback:
  + If you need to check one element back
  + If you need to check a constant number of elements back
  + If your restriction moves forward with 1
  + If you need to carry the max, sum, count, etc. forward
    - Since you carry your answer forward, you only compare against i – 1
  + This results in O(n)

**Longest Common Subsequence**

1. If your problem is comparing between two arrays
2. If your problem is looking for something in common
3. This is normally (there are exceptions) a 2-D table
4. There are two variations:

* Substring:
  + If you want the matches to be consecutive
  + Add 1 (or whatever) if the comparison succeeds
  + 1 + T(i – 1, j – 1)
  + Reset to 0 when a comparison fails
  + The solution is the max of T()
  + This results in O(n\*m) or O(n2) depending on if the two arrays have different sizes or not
* Subsequence:
  + If you want the comparison to be able to skip mismatches
  + Add 1 (or whatever) if the comparison succeeds
  + 1 + T(i – 1, j – 1)
  + Take the max of either side if the comparison fails
    - max {T(i, j – 1), T(i – 1, j)}
  + The solution is the final results T(n, m) or T(n, n)
  + This results in O(n\*m) or O(n2) depending on if the two arrays have different sizes or not

**Edit Distance**

1. This is very similar to LCS and considered a 3rd variation
2. If your problem is comparing two arrays
3. If your problem is looking to minimize differences
4. If your problem assigns some penalty to differences
5. If your problem assigns some points to matches
6. If you are aligning two arrays
7. This is normally (there are exceptions) a 2-D table

For i = 1 to n

For j = 1 to m

T(i, j) = min/max {T(i – 1, j – 1) + function(i, j),

T(i – 1, j) + function(i, j),

T(i , j – 1) + function(i, j)}

1. Basically, you declare a function to determine the value, penalty, etc. of aligning i and j
2. Then you find the optimal position of i and j by checking the 3 possible ways to align them
3. This results in O(n\*m) or O(n2) depending on if the two arrays have different sizes or not

**Knapsack**

1. If your problem needs to check if something can fit or add up to
2. If your problem needs a maximum value for a specific budget
3. If your problem needs to find the minimum budget needed for a specific value
4. Two variations:

* Limited items:
  + This is normally (there are exceptions) a 2-D table
  + For each increment of the budget
  + For each increment of the items
  + Checks if the item can fit – if w[i] <b
  + Results in O(nB)
* Unlimited items:
  + This is normally (there are exceptions) a 1-D table
  + Set the initial values of each b to 0
  + For each increment of the budget
  + For each increment of the items
  + Checks if the item can fit – if w[i] <b
  + Does not need to track if “i” was used
  + Results in O(nB)

**Windowing (Chain multiply)**

1. If your problem needs a windowing check
2. If your problem is asking for the most efficient way to split an array
3. If your problem is asking if an array can be split in some way to do something
4. This is normally (there are exceptions) a 2-D table
5. Two variations:

* Windowing only:
  + Set a window size of 1 to n
  + Set i = 1 to n – window
  + Set j to i + window
  + Calculate T(i, j) using smaller windows between i and j
  + You can do this because your window is growing, so all smaller windows T(i, j – 1) and (i + 1, j) are already calculated
  + This results in O(n2)
* Windowing and break at midpoint:
  + Set a window size of 1 to n
  + Set i to 1 to n – window
  + Set j to i + window
  + Set k (break) from i + 1 to j – 1
  + Track (not in table) the max of T(i, j) by moving k and finding what the optimal position for k is to maximize T(i, j)
  + You can do this because your window is growing, so all smaller windows T(i, k) and T(k + 1, j) are already calculated
  + This results in O(n3)

**Graphs (Bellman-Ford and Floyd-Warshall)**

1. Understand how they work in black box (4.11, 4.21)
2. Understand what they solve, their requirements, what they can't solve, and runtimes